An application Framework for Collaborative, Nomadic Applications

James O'Brien, Marc Shapiro

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

An Application Framework for Collaborative, Nomadic Applications

James O'Brien, Marc Shapiro

N° 5745

Octobre 2005

Thème COM

Rapport de recherche

INRIA
ROCQUENCOURT

An Application Framework for Collaborative, Nomadic Applications

James O'Brien[1], Marc Shapiro[2]

Thème COM – Réseaux et systèmes
Projet Regal

Rapport de recherche n°5745 – Octobre 2005 - 26 pages

Abstract: To maintain availability and responsiveness, mobile applications sharing data often work on their own copy and transmit their local changes to other participants. Existing systems for recording, transmitting and reconciling concurrent changes are usually ad-hoc and specific to particular applications. In contrast, we present Joyce, a general application programming framework for creating highly dynamic mobile, collaborative applications. The framework abstracts application semantics using an action-constraint formal model and provides communication and consistency services based on this model. The framework exposes an interface that allows application programmers to concentrate on core functionality without worrying about these issues. Applications made with the framework can run seamlessly across changing combinations of devices, users and synchrony. We discuss the principles behind the framework, its implementation and evaluate its utility by implementing a complex, shared application.

Keywords: CSCW, multi-synchronous collaboration, selective undo/redo, concurrency control

[1] www.jaimz.org, james@jaimz.org

[2] INRIA Rocquencourt, France and LIP6, Paris, France, http://www-sor.inria.fr/~shapiro/

Unité de recherche INRIA Rocquencourt

Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Un canevas applicatif pour les applications collaboratives et nomades

Résumé: Afin de rester toujours disponibles, les applications mobiles partageant de l'information travaillent souvent sur une copie privée et transmettent leurs modifications locales aux autres participants. Les systèmes existants, visant à l'enregistrement, la transmission et la réconciliation des mises à jour concurrentes, sont souvent *ad-hoc* et spécifiques à une application particulière. Au contraire, Joyce, le canevas logiciel pour applications collaboratives et mobiles hautement dynamiques, présenté ici, est générique. Le canevas est basé sur un modèle abstrait de la sémantique applicative utilisant le formalisme actions-contraintes, et propose des services de communication et de maintien de la cohérence, basés sur ce modèle. L'interface du canevas permet aux programmeurs d'application de se concentrer sur la fonctionnalité métier sans avoir à se soucier de propagation ou de cohérence. Les applications qui s'appuient sur notre canevas s'exécutent sans heurts sur des combinaisons variables de machines, d'utilisateurs et de synchronicité. Nous détaillons les principes qui sous-tendent le canevas, sa mise en œuvre, et nous évaluons son utilité par la réalisation d'une application partagée complexe.

Mots clés: CSCW, travail coopératif aidé par ordinateur, collaboration multi-synchrone, défaire-refaire sélectif, contrôle de concurrence.

• Introduction

It has become clear that there is an increasing demand for applications to work in an environment that is both mobile and collaborative; users are running them on devices such as laptops and PDAs that are not geographically fixed and are using them in concert with other users to concurrently edit shared data. Web applications are also starting to exhibit this trend with popular tools such as Wiki [Wiki] and JotSpot [JotSpot] providing rudimentary collaborative facilities over the web. The environment that emerges is characterised by a high degree of change in the number of participants, change in connectivity between those participants, and change in the synchrony of collaboration - collaborators may be sitting next to each other or in different time zones. Programmers need tools to create good collaborative, nomadic applications: applications that adapt to mobility, adopt a collaborative posture but retain the richness and control of desktop applications.

The major problem with such applications is maintaining the consistency of shared data. Most of the commonly used application architecture models, for example Model-View-Controller [Krasner 88], contain an implicit assumption that application data is modified by one user using one device. Many applications which would benefit from collaboration and mobility do not due to the prohibitive cost of architecting the application to take account of concurrency control issues.

Certain classes of application, usually PIM applications, are designed specifically to be shared between mobile devices. The techniques used however, are specific to the data domain of the application and intrusive to the application logic. The application developers must dedicate a lot of time and effort to concurrency issues, neglecting the application's core functionality. Moreover, these applications usually use some kind of lock-step synchronisation which requires the user's intervention. Finally, the concurrency control wheel tends to be reinvented with each application, extending development time and resulting in segregated, incompatible systems. This is not an approach that scales well to general application construction and the increasing popularity of pervasive, mobile computing is likely to underscore its shortcomings.

Functionality time-consuming to implement but common between different applications is usually encapsulated in an *application framework*. An application framework is designed to handle the logic common to all applications sharing a particular aspect: for example Apple's Cocoa framework [Cocoa] handles interaction with the windowing system for graphical desktop applications. Frameworks differ from libraries in that applications using them exhibit an *inversion of control* [Schmidt 00]; it is the framework logic, rather than the application logic that controls the execution of the application process.

In this paper we describe an application framework called Joyce that introduces a new programming pattern for highly dynamic collaborative applications and provides an implementation of that pattern. Joyce is based around an *optimistic replication* system that enables applications made with the framework to run across changing combinations of devices, changing combinations of users, and changing combinations of synchrony. The framework exposes a programming interface that allows the application programmer to concentrate on core functionality without worrying about these issues. We describe what we believe are the current and future requirements of collaborative, nomadic applications and why current techniques do not meet these requirements, we then go on to explain the principles behind our system and describe a realistic application, "Babble", created to evaluate the system.

• Requirements

Applications created with our framework must meet the following expectations:

- *We expect to be mobile and only occasionally connected*: the applications will be used concurrently by a mixture of users on a mixture of devices. Devices may transition between on-line and off-line at any time so we cannot assume constant connectivity or a complete knowledge of the collaborative group membership. We also cannot assume any particular physical device configuration (e.g. local storage).
- *We expect nomadic, collaborative applications to be as rich as current single-user, single-device applications*: the applications must be at least as responsive and featureful as current desktop applications and will preferably exhibit improvements in usability.
- *We expect to be fully aware of group activity but we do not expect to be bound to a distracting WYSIWIS environment*: these environments (**W**hat **Y**ou **S**ee **I**s **W**hat **I** **S**ee) attempt to keep the application display of each participant precisely in sync. Where such a scheme is necessary (most often in conferencing applications such as shared whiteboards) we expect the framework to allow us to build it. However, in applications where real-time collaboration is not the objective, the immediate execution of incoming modifications produces a display that constantly distracts the user from his local task. This leads to a feeling of loss of control which in turn leads to application usability far lower than the single-user equivalent; which as we have already stated is unacceptable. We expect to be continuously *aware* of group activity but also in *control* of how and when the activity is applied.
- *We expect to be aware of the group history of the application state and we expect a manipulatable history that works well in collaborative environments*: projects such as FlatLand [Edwards et al. 00] and GINA [Berlage et al. 93] have demonstrated the benefits of manipulatable history but current implementations of undo/redo in a collaborative environment are complex and application specific. [Sun 02]

To meet these expectations and remain generic the framework needs to be adaptable across two major criteria. Firstly, the framework must be able to cope with different degrees of *coupling* between the participants [Berlage et. al. 93]. Coupling is the degree of co-ordination between participants. For example, when syncing mobile devices all the devices involved are connected and they all receive each other's updates at the same time. In contrast, collaborative systems can fall anywhere between same place/same time systems where collaborators work “shoulder-to-shoulder”, to different place/different time systems where collaborators may be dispersed across time zones. We should be able to use the framework to build applications anywhere within this spectrum.

Secondly, any concurrency control system is closely linked to the semantics of the object being shared [Munson et al. 96]. In traditional database systems this semantic is one of read/write operations to some storage. This was found to be too restrictive for many collaborative systems and techniques were developed to expose a richer set of semantics based on the programmatic interface of the shared data structures [Munson et. al. 96][Schwarz et. al. 84]. It has been demonstrated that these systems allow more concurrent activity since they can more narrowly define what constitutes a conflict and thus maximise the number of concurrent operations that can be run on a shared state. From a user's perspective however, a modification has more semantics than can be expressed solely in data structure interfaces and a good concurrent application will also take into account user intentions and higher-level application semantics. Our framework must provide an application agnostic method of capturing the full semantics of a modification, both object and user-level.

1. Problems to solve

From these general requirements we developed a more concrete list of problems to be moved from the domain of the application to the domain of our framework:

- **Modeling activity**: Joyce needs to provide a way of describing modifications that is generic enough to model any application. Further, Joyce should also provide a generic way to represent concurrency semantics that is rich enough to articulate object, application and user-level semantics.
- **Communicating activity**: The framework should ensure that, despite changing connectivity, modifications from one participant will propagate to all the others.
- **Reconciliation**: Replicas may be independently modified in a conflicting manner. Reconciliation is the process of bringing replicas to a consistent state by detecting and resolving conflicting concurrent modifications. The process of detecting and resolving conflicts depends on application semantics and user intent and existing reconcilers [Balasubramaniam & Pierce 1998] are confined to a single data type. Joyce remains application agnostic by representing application semantics and user intents explicitly.
- **Consistency**: Individual replicated states are allowed to diverge in the short term. However, some applications require that all states must eventually be made consistent according to the results of reconciliation. Joyce has a mechanism for bringing a state to consistency, concurrent with the user modifying that state.

In satisfying these problems it is vital that Joyce not degrade the performance and responsiveness of the application. This is especially important when transitioning from connected to disconnected states, transitioning between asynchronous and synchronous collaborative modes, and bringing a state to consistency.

2. Previous Work

An early approach to concurrency control was simply to acquire a lock on a piece of data before modifying it, the data being stored at some central location. If the lock could not be acquired then the application was allowed to modify the data and either blocked until the lock was available or failed. Many early research systems were based around a locking mechanism called floor-control [Sarin et al. 85] in which one participant modified the shared object while the others observed, waiting their turn. This approach has the advantage of simplicity and is still used in current web-based systems such as Wiki [Wiki]. However, locking has proven problematic for mobile applications since it requires a constant connection to the central data store, and even if a connection is present an application may spend a great deal of time blocked until a lock becomes available.

The DistView [Prakash et al. 94] framework used replicated lock tables to prevent blocking becoming too great a hindrance and the GroupKit [Roseman et al. 96] system allowed operations on shared data whilst a lock was pending; if the lock request was refused the operations were undone. The concept of *tickle locks* [Greif et al. 86] was developed to minimise the amount of time waiting on a lock - essentially the requester would 'tickle' the participant holding the lock and, if there was no response, the lock would be transferred.

Even with these improvements, locking proved restrictive and lead to awkward interaction as applications either blocked or backed-out failed changes. Instead, mobile applications often adopt an optimistic replication scheme [Saito et al. 05] in which each participant takes a local replica of the shared state and modifies that replica without regard to concurrent changes from other applications. At some later point all the replicas are synchronised to produce a common state. The technique is termed optimistic since the applications 'optimistically' assume that their local changes will not conflict with concurrent changes at other replicas. This is the approach used in our framework since local states require no locking and the applications can remain responsive.

The dOPT algorithm of Ellis and Gibbs [Ellis et al. 89] introduced to notion of operational transform (OT) in which an application receives an operation issued remotely and re-writes it such that its effect is the same locally as it was where it was issued, regardless of any local operations that have happened in the mean time. OT has proven particularly popular in real-time collaborative text editing systems such as ShrEdit [McGuffin et al 92], Grove [Ellis et al 88] and SubEtherEdit [SubEtherEdit].

The use of OT leads to very responsive applications but the technique is more a mechanism to maintain consistency despite out-of-order messaging than a synchronisation mechanism. Moreover, although the technique itself is generic, OT implementations are usually application specific and very complex. The semantics of an operation is obfuscated by the transform and often lost entirely if an incoming operation has to be transformed against many prior operations. If a history mechanism (such as undo/redo) is required this leads to further application-specific complexity [Sun 02]. There are also known scenarios where current OT techniques may lead to an inconsistent state [Li 04]. Finally, OT is intended primarily for real-time, synchronous editing systems rather than multi-synchronous, occasionally-connected systems.

Bayou [Edwards 97] introduced several mechanisms that support multi-synchronous distributed applications. Bayou is a log-based optimistic replication system that models operations using a read/write semantic augmented with application-defined conflict detection and resolution mechanisms. Operations are communicated using an epidemic propagation scheme that guarantees updates from one participant will reach all the others given sufficient connectivity [Demers 87]. Bayou has good solutions for maintaining communication in the face of occasional connectivity but forces applications to adhere to the limited read/write semantic.

Although concurrency control has been studied extensively and many techniques have been developed we find none of the principles and algorithms suitable to be integrated into the general application development cycle. Either the techniques are too complex or application specific (as with OT), do not work in a multi-synchronous environment (as with floor-control) or do not wholly express application and user semantics (as with Bayou).

• System Overview

We cannot fully automate concurrency control since resolving conflicting concurrent actions depends on the application semantics and user intents. Nevertheless, we are able to create a general programming framework by basing Joyce on a reified model of application semantics. In this section, we present a high-level view of how the system works and the core concepts used.

1. Basic Model

Joyce is a programming framework that offers an operation-based replication and collaboration system. Joyce connects participants that are working on the same shared data and distributes the modifications made by one participant to all the others. It allows participants to disconnect and reconnect without loss of information or responsiveness; an application can continue to run while disconnected and modifications will be propagated to it on reconnection.

We define a *data object* as the distinguishable unit of data that is being shared, this may be anything from a calendar to a document to a database. Each data object has an associated *group* which is the notional set of all nodes working on replicated copies of that data object; a node being some application process that is modifying the data. Note that it is possible that the members of the group may change from one moment to the next as may the connectivity between members. We cannot require that any member have a complete knowledge of all the others but we do provide a mechanism that any one node can use to discover a *peer group* – the subset of the group that can be contacted. The framework ascertains the peer group either by broadcasting an announcement and listening for replies or by joining an application-level multicast tree [Castro et al. 02] corresponding to the shared object.

2. Modeling Application Activity

Following the command pattern [Gamma et. al. 95] Joyce applications are architected as a set of commands that modify a particular kind of data object. Unlike the command pattern, command invocations in Joyce are augmented with a set of *constraints* designed to explicitly articulate the intended application and user semantics. Each command invoked is recorded in a log along with constraints describing the semantics of the modification the command is part of.

The log entry describing a command invocation is called an *action* and is designed to allow the corresponding command invocation to be reproduced at another member of the group. It consists of a set of attributes including the source node that executed the command, the application that contains the command, a unique identifier for the command and the parameters of the command.

A constraint is designed to represent a semantic invariant of the application. The system is entrusted to maintain these invariants when reconciling concurrent modifications. User intents and application semantics are recorded using a category of constraint called *log constraints*; this is in contrast with previous systems, where only the chronological order of operations is recorded [Petersen et al. 1997]. The concurrency semantics of the shared objects themselves constitute a separate sub-category of constraint called *object constraints*. Joyce uses the set of log and object constraints defined by [Preguiça et al. 2003a], explained next.

Figure 1 Joyce logs modifications with their semantics. In Babble, a search and replace is logged as insert and delete actions that are ordered and atomic.

1. Object Constraints

Object constraints represent the semantic invariants between pairs of concurrent actions, and via these invariants the concurrency semantics of the shared object modified by the actions. Object constraints are defined using the following set of pair-wise relations:

- **Commutes:** Do the supplied actions commute? Is the result of executing the two actions independent of execution order?
- **Helps:** Does running the first action before the second increase the chances of the second succeeding?
- **Hinders:** Does running the first action before the second decrease the chances of the second succeeding?
- **Enables:** Can the second action be run only if the first action has succeeded?
- **Prevents:** Does running the first action prevent the second action from succeeding?

2. Log Constraints

Log constraints express invariants that must hold between action instances that share a log (as opposed to object constraints that express invariants between *classes* of action). We have factored log constraints into two categories: *grouping* constraints and *ordering* constraints. Currently there are two types of grouping:

- **A parceled grouping**: Confers atomicity to the grouped actions. Either all the actions must be executed or none of them can be.
- **An alternative grouping**: Indicates that only one of the grouped actions can be executed.

Ordering constraints indicate that the constrained actions should execute in the specified order. Following [Preguiça et al. 2003]:

- **Strong ordering** indicates that, if the predecessor cannot be executed, then neither can the successor.
- **Weak ordering** indicates that if the successor has already executed, then the predecessor may *not* (but the other way around is OK).

Log constraints are typically used to express higher level application semantics (i.e. a subtask within an application that consists of more than one command) or user intents within a particular task. For example, Babble uses the parcel constraint to make a *search-and-replace*, composed of multiple deletes and inserts, into a single semantic unit.

3. Modeling Group Activity

An application's local log forms part of a larger data-structure called the *multi-log*.
Whereas a log represents activity at a single node a multi-log represents, as best we can, application activity across the whole group by keeping a log entry for each member that has ever been in a node's peer group.

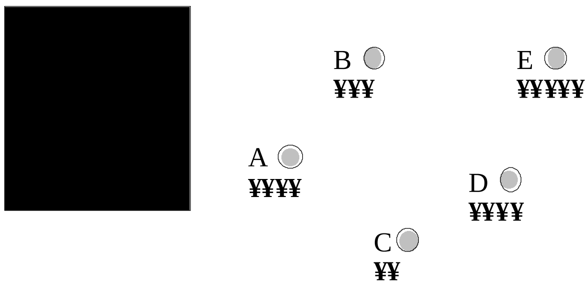

Figure 2 A five node Joyce group and its multi-log representation on A.

Figure 2 shows a five node Joyce group. Each node (represented by a grey circle) is shown with an indication of the activity that has occurred on that node. Also shown is A's knowledge of that activity as captured in the multi-log on A. This multi-log has accurate records for B and C but is out of sync with regard to D and E. It is a key task of the framework to keep each multi-log on each node as representative of the group activity as possible.

Log constraints can be placed between the individual logs of a multi-log to create a *semantic graph* that describes the activity within the collaborative group. For example, if a modification on one node depends on a modification on another a strong ordering constraint will be placed between the respective logs in the multi-log. In this way we build up a picture of the activity within a group that is independent of the chronology of the actions. Instead of trying to use timestamps to derive dependency information we use the invariants expressed in the multi-log semantic graph.

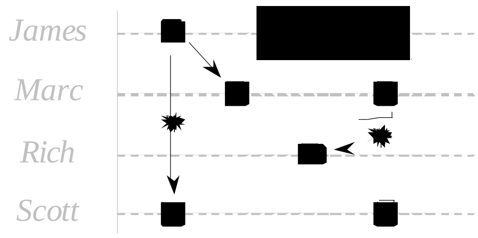

Figure 3 This multi-log describes a semantic graph containing an ordering constraint, two conflicts and a parcel.

4. Propagation

In an occasionally-connected group, it is impossible to keep the multi-logs perfectly representative at all times. However, the framework should exhibit the property of *eventual consistency* which states that when the group is quiescent (no membership changes and no new updates) all nodes should eventually receive all updates within the group given sufficient connectivity.

To achieve this we use an epidemic propagation scheme [Demers 87]. Joyce logs are formatted as a linear list of *elements* and each element appended to the log is assigned a monotonically increasing sequence number - its index in the log. There are currently two kinds of log element: actions and meta-actions.

Figure 4 Logs are kept as a list of actions and meta-actions. Each element in the list is assigned a sequence number.

Action elements are representations of the action construct described in section 3.2. Meta-actions, in contrast, specify information about actions prior to them in the log (i.e. actions with a lesser sequence number.) Unlike actions, meta-actions are not functions of the application but are used to supply information about actions to the Joyce system, most usually constraint information. Meta-actions can be thought of as instructions on how to build the semantic graph that the log describes. This scheme allows us to capture both modification and semantic data in a sequential collection of elements.

Figure 5 The *Parcel* meta-action indicates that the actions at sequence numbers 0 and 1 should be executed atomically. It has placed a *parcel* constraint.

Epidemic propagation distributes multi-log updates by making a series of pair-wise exchanges between connected peers in the group. These exchanges may be prompted by a connectivity change (for example when a member joins a group), they may be timed to occur at certain intervals or they may happen during an interaction pause (i.e. if no new actions have been logged after a certain period).

During the exchange each peer transmits a vector clock indicating how fresh its multi-log is. The receiving peer subtracts the incoming vector clock from its own vector clock to see if the source peer has fresher updates for any logs. If it does then the updates are requested.

Epidemic propagation exhibits good behaviour in the face of varying connectivity since a node's updates may still propagate through intermediaries even if that node is no longer connected [Demers 87].

5. Reconciliation

Epidemic propagation is the mechanism for keeping the multi-logs in sync, reconciliation is the mechanism for keeping the replicated states in sync. Reconciliation is the act of merging the concurrent logs from each node into a schedule containing only non-conflicting actions.

Most existing reconcilers merge according to a pre-determined order (for example timestamp order in Bayou). In contrast, Joyce builds on our previous reconciliation system IceCube [Preguiça et al. 2003a], which uses the constraint information to produce a schedule containing as many actions as possible whilst still preserving application semantics.

IceCube treats reconciliation as an optimisation problem. Recall that the multi-log is a graph in which the concurrent actions are the nodes and the constraints between them the edges. A schedule is a traversal of this graph such that all the static constraints are satisfied. Any actions not traversed are dropped. If a dynamic invariant fails, the scheduler backtracks and tries a different traversal. The scheduling heuristic we have developed finds a traversal that minimises the value of the dropped actions.

The reconciliation algorithm may suggest many possible schedules when presented with the same input but only one will be selected for *commitment*. Commitment is the act of irrevocably selecting a reconciliation schedule for execution at every member in order to make their replicated states consistent. The schedules that have been committed are recorded in a special multi-log entry called the *commit log*. The commit log consists of commit and abort meta-actions referencing actions in the multi-log that have been committed (irrevocably scheduled for execution) or aborted (irrevocably excluded from execution).

Schedules are generated by a reconciliation engine that implements an incremental version of the IceCube algorithm [Kermarrec 01]. If the user is satisfied with a schedule it is appended to the commit log and propagated with the rest of the multi-log.

A node that generates commit-log updates is called a *primary* and there is usually only one per Joyce group. Epidemic propagation ensures that the commit log updates generated by the primary propagate to all the other nodes in the right order and eventual consistency is reached.

6. Storage and Checkpoints

The traditional file system storage model is cumbersome when applied to nomadic, collaborative applications. Nomadic devices may not have local storage and continuous connection to a file server is not feasible. Moreover, an important philosophy of Joyce is that editing his data should be the user's main, preferably his only, focus of attention. To this end the framework provides an automatic persistence service that requires little user intervention.

One or more entities called *storage nodes* can be configured to join a collaborative group. These storage nodes are peers that consume the traffic flowing through a group and persist the generated multi-log to backing store.

Storage nodes may serve several purposes: a storage node that is permanently online may replace a central file server; or devices with appropriate resources may run a local storage node that persists the multi-logs of all applications running on the device.

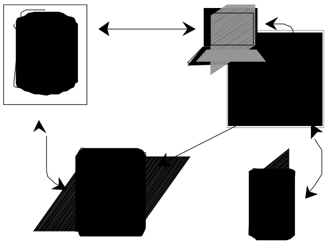

Figure 6 A Joyce group containing two storage nodes: a 'server' node and a laptop running its own node.

Joyce applications take snapshots of their data at specific times. A snapshot is a binary image of the application's current state annotated with the vector clock of the multi-log when the shot was taken plus whatever other descriptive meta-data is provided.

A snapshot is most often taken when a state has reached some milestone in the editing process or when the framework detects that the state has been brought to consistency (by an application of the commit log). Taking a snapshot of the consistent state allows the framework to truncate the multi-log by removing the committed and aborted actions - all future actions can be issued against the consistent snapshot. Snapshots are automatically stored and managed by storage nodes and applications may roll-back to prior snapshots.

If a group member disconnects or crashes the act of re-joining the group, and re-contacting the storage node, restores the application state with a minimum of data loss. On re-connection, an application can either replay the entire multi-log or load an appropriate snapshot and replay the sub-logs subsequent to that snapshot. See section 5.6.2 for a practical example of using snapshots.

• Application Model

In section 3 we described the low-level communication, consistency and storage systems in Joyce. In this section we will explain how we create applications that use those systems.

Joyce provides a skeleton application architecture designed to foster applications that meet the expectations in section 2. The key principle of the architecture is that the user interacts with a *local view* of the global activity which is as responsive as a corresponding single-user application would be. The user should feel in full control of this local view and not overwhelmed by group activity.

The architecture keeps this local view responsive by applying local modifications immediately, implementing undo/redo as local operations that are group aware, and by filtering the multi-log to tailor how much group activity is shown. To use the architecture, client programmers supply a set of actions and constraints that encapsulate their application’s functionality and a base state against which the actions can be run. The architecture will take care of propagation, reconciliation, storage and execution of the actions against the base state.

1. The Active Subset

The architecture is based around an *active subset* - a subset of actions from the multi-log that is run on the base state to generate the local view. The subset provides a uniform way to communicate both group activity and history manipulation (e.g. undo) to the application. It is by configuring which actions to include in this subset that the user can control what appears in his application display.

The subset contains two kinds of action: actions that have been committed by the primary and a consistent subset of *tentative* actions - actions that have not yet been committed or aborted.

A subset is consistent if all the constraints in the subset are satisfied and none of the actions conflict. For clarity of explanation we will assume that the base state the active set applies to is the initial state (the state before modification), see section 3.6 for details on how Joyce can truncate a multi-log by making a snapshot of a stable state.

2. The Local Interaction Cycle

To reflect local modifications quickly, the architecture populates the active set using an *interaction cycle* derived from the Model-View-Controller pattern [Krasner 88]. An interaction cycle is the programmatic path between a user triggering a local modification and the result of that modification being reflected in the application output. MVC introduced a cycle, depicted in figure 7, in which input from the user is evaluated by a controller into a set of modification messages for the model; the model applies the modifications and sends a set of update messages to the view which reflects the model change back to the user.

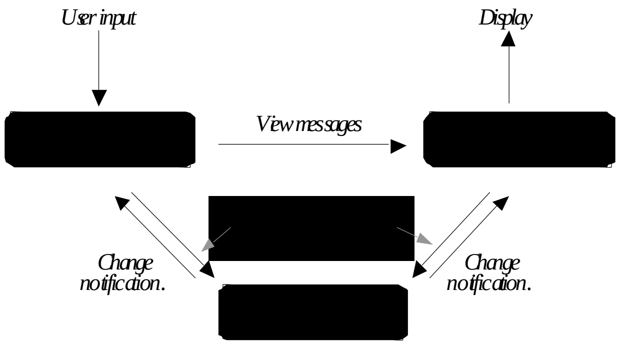

Figure 7 The traditional MVC interaction cycle.

This pattern simplifies the construction of GUI applications but assumes that modifications always come from a local (i.e. in-process) controller; and inversely that modifications from the controller are always for the local model. The pattern also has the more subtle assumption that the local controller is the authoritative source of the modifications - it has no notion of a global state that might be defined elsewhere.

We expand MVC by introducing another component, the *coordinator*, whose job is to maintain the active subset and apply it to the model. During our interaction cycle (figure 8) user input is evaluated into a set of actions and constraints representing an application command; these are sent to the coordinator, which logs them in the multi-log and immediately includes them in the active subset - causing them to be applied to the model and reflected in the view. We call this the *local interaction cycle* since the actions applied to the state are local, tentative actions.

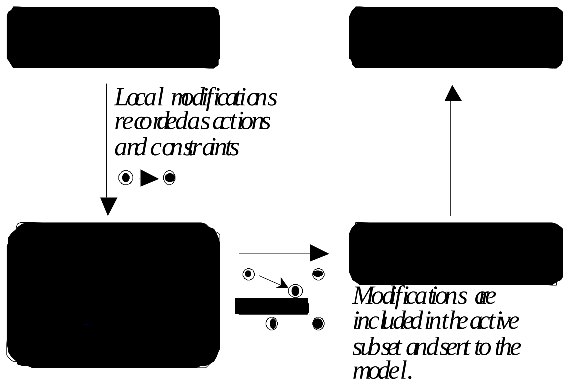

Figure 8 The interaction cycle in Joyce. The controller generates modifications and sends them to the coordinator for execution and logging.

When an update to the multi-log arrives from a peer, the coordinator interrupts this cycle to re-calculate the active set. It uses the reconciler to create a consistent schedule from the pool of tentative actions in the multi-log, including the new arrivals and the local actions generated by the cycle above. This schedule becomes the new active subset.

Applications may influence which actions are included in the schedule by assigning weights to specific actions to indicate greater importance. For example, Babble specifies that local actions take precedence over remote ones by ensuring that they have a greater weight; this prevents local edits disappearing when a multi-log update arrives and is expected to be a common technique in Joyce applications.

The reconciliation that occurs when a multi-log update arrives has no effect on the globally consistent state defined by the commit log - it is local to the receiving node. If the multi-log update includes a commit log update the aborted actions and their dependants are removed from the tentative action pool and the active subset is pre-populated with the committed actions before the local reconciliation occurs.

The actions in the active subset are recorded relative to the base state. To apply a new active subset, Joyce restores the base state, using either a stored checkpoint (above) or compensation actions, then runs the new active subset against it. The schedule produced by the reconciler is guaranteed to respect ordering constraints and so can be executed sequentially.

3. Undo, Redo and Filtering

A user can manipulate his active subset by placing filters over the set of tentative actions in the multi-log. A filter is simply a predicate that pre-excludes matching tentative actions from a reconciled schedule. This prevents the coordinator including the filtered action and its dependants in the active subset.

The simplest example of filtering is masking out specific collaborators. Here, the filter matches every action with a particular author. Actions from the author will not be accepted into the active subset and thus will not contribute to the local state. It is important to note that filtering does not remove actions from the multi-log, just from the tentative action set. All information about group activity is retained, an important expectation (section 2). Later, the filter may be removed, allowing the previously masked work to be reintegrated into the view.

Undo is implemented as a filter that masks out a specific action. To undo a modification the user selects the action and creates the filter; the tentative actions are then re-reconciled and the resulting active subset will be equal to the previous active subset less the undone action and its dependants (those actions that are parceled with or strong ordered after it). If subsequent remote actions arrive that are dependant on the undone action the reconciliation process ensures those actions will not appear in the active subset.

Since constraint information is used to calculate the dependants, undo in Joyce is *selective.* The undo operation is confined only to those operations directly effected [O'Brien 04] and the corresponding redo can be done at any time if no intermediate arrivals conflict with the undone action. This is in contrast with the stack-like, linear model used in most applications.

• An Example Application: Babble

To refine Joyce for real-world development we created a collaborative text editor named 'Babble'A text editor is complex enough to exercise the whole framework but is familiar enough that the contributions of the framework are well highlighted: particularly collaboration, selective undo/redo and passive storage.

1. Representing Text Editing to Joyce

Applications built on Joyce are required to provide a collection of core operations (the actions) and the concurrency semantics of those operations (the constraints). We must therefore formulate a set of text editing operations and describe, using Joyce constraints, the invariants between them.

Text editors are usually modeled using a linear text buffer that is addressed using character position co-ordinates from 0 (before the first character) to N (the position after the last character). Two operations are defined: *insert*(p, c) that inserts character c at position p and *delete*(p, n) that removes n characters starting at position p. Concurrent text editors have traditionally used the same model and have employed operational transforms [Ellis & Gibbs] to 'correct' remote insert and delete operations received out of order. OT gives good performance in distributed, real-time editing but leads to considerable complexity if multi-synchrony and undo-redo are required [Sun 02], intrinsic qualities of Joyce applications.

Babble uses a more systematic approach that meets the requirements of the Joyce framework. The approach is based on a more expressive representation of a text buffer that captures dependencies between editing operations and allows us to show, hide, recombine and re-order operations as directed. The representation is in three parts:

- A linear text buffer, the *content*, similar to the structure used by non-concurrent and OT editors. However, with the exception of snapshots and undo (see below) characters are only ever inserted into the buffer, not removed.
- A collection of character position intervals, the *mask*, that indicates text that has been deleted. Babble's display logic ensures that masked text is not displayed and therefore cannot be subsequently edited (the cursor cannot be placed in the masked content).
- A hierarchical collection of character position intervals, the *history*, that records the operations that have been applied to the content in terms of the range of characters effected. It is from this we calculate the Joyce constraints.

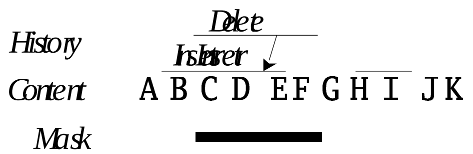

Figure 9 Babble represents a text buffer in three layers. This buffer is displayed as ABGHIJK

Babble defines two actions: *Insert*(p, s) that inserts the *string* s into the content at character position p and *Delete*(p, n) that creates a mask of length n starting at character position p. Each operation executed results in a new interval in the history. For example, starting with the text:

```
A B C D E F G
```

The sequence *Insert(2, 'pqr')* and *Delete(3, 3)*, produces the data structures:

```
history:        |-I---| |-D---|
content:     A B p q r C D E F G
mask:                    -----
```

Which will display as:

ABpqrCG

Dependencies are expressed using the Joyce *Strong Order* constraint (see section 3.2.2). The history structure is used to calculate the dependencies to log. For *Insert* operations we say that one *Insert* α depends on another *Insert* β if, within the history, the edit point of α falls within the span of β. For *Delete* operations we say that a *Delete* α depends on a *Delete or Insert* β if the spans of α and β intersect. Following from the previous example, if an insertion is made between p & q and the range between u and G deleted, the history will be:

```
         |-D2------------|
          ↓             |
     |-I2--|            |
        ↓               ↓
   |-I1--------| |-D1--|
A B p s t u q r C D E F G
```

That is, D2 depends on both I2 and D1. There is no need to place an order constraint to I1 since the constraint to I2 is transitive.

2. Responding to the Active Subset

Joyce applications running on separate nodes individually select and apply a consistent subset of the global history: the active subset. This is the mechanism through which Joyce uniformly represents history modifications and collaborative activity. Babble should react to active subset changes by dynamically changing the displayed content to reflect the subset. This requires Babble to be able to replay local and remote operations in any order, since they may be recombined in any consistent order by the Joyce reconciler when the active subset is calculated.

Most concurrent editors use OT to replay out of order actions from remote sites. Essentially, OT translates the character position of inserts and the character position and spans of deletes from an action's original state into the current local state. The technique used in Babble is derived from OT in that it uses the same idea of translating edit points; however, our technique allows us to retain a systematic operational history and is far simpler to implement than pure OT approaches.

Each content insertion creates a *scope* consisting of a 1D co-ordinate space from 0 to the number of characters inserted. Each subsequent *Insert* and *Delete* operation applied to a scope is recorded in this original co-ordinate space regardless of the mutations caused by prior operations. When an existing scope is intersected by an *Insert* operation we keep a record of the *shift* introduced by the inserted content. For example, a document with the initial content `ABCDEFGH` would have the scope:

```
 A B C D E F G H
|---------------|
0       +0       8
```

That is, a scope from 0 to 8 across which there is a shift of 0. If an insertion of `pqr` is made between `B` and `C` the following scope is created:

```
      I(2, 'pqr')
    |-----|
 A B p q r C D E F G H
|---       -----------|
0   |-----|     +3    11
      +0
```

A new scope corresponding to the insertion has been created and the intersected scope has been split, with the portion after the insertion weighted according to the characters inserted. The weight is used to map subsequent edits into the original coordinate space:

```
   I(2, ' pqr' )    I((8-0-3), ' stu' )
                    = I(5, ' stu' )
      |--------|          |--------|
 A  B  p  q  r  C  D  E  s  t  u  F  G  H
|-----         --------        -------|
0    |--------|    +3   |--------|  +6   14
        +0                 +0
```

In this way, all operations in a scope are recorded in the original co-ordinate space of that scope and can be replayed in any order. This process is recursive, for example if an insertion is made between `q` & `r`:

```
    I( (4-2-0), 'wxy') = I(2, 'wxy')
        |-----|
           ↓
      |-------------|      |-----|
   A B p q w x y z r C D E s t u F G H
0 |---               -----      ------| 14
   +0|---       ---|  +3 |----|  +6
       +0 |-----| +3          +0
     +0
```

The strong order constraint ensures that the correct scope is in place before a dependant action is replayed.

Scopes allow us to map the original edit point of an operation into the current state. If a remote peer inserts ‘wxy’ between C and D in a replica of the initial document then the incoming action will be *Insert(4, ‘mno’)* with no order constraint. When replaying, Babble detects that the intersected scope in the local replica is not correct and bumps the edit point until the correct scope is reached (in this case the initial scope).

```
              I(4, 'mno')
                = I(10, 'mno') after transform
                |-----|
 A B p q w ... C m n o D E s t u F G H
|---           -       ---       -----|
    |---        |-----|   |-----|
           |- ...
      =========>
       Edit point of the incoming action is shifted until it reaches the correct
scope.
```

The mask structure allows us to use the same scheme for *Deletes.*

Our replay strategy does not suffer from the same concurrency 'puzzles' as a linear buffer using OT. For example, when presented with the TP2 puzzle outlined in [Li et al. 04], our structure exhibits convergence *and* intention preservation unlike most OT approaches.

Babble implements the policy that if the spans of concurrent actions overlap then those actions are in conflict. This is expressed to Joyce using the mutual exclusivity static constraint and Joyce will notify Babble if any action in the active subset conflicts with another action in the multi-log. Babble will not try to automatically resolve the conflict but will highlight the content contributed by the conflicting action with a red shading in the display (see *User Experience* below).

3. User Experience

In this section we give a brief overview of the user experience in Babble and detail our preliminary ideas about how to communicate Joyce concepts to the user in a complex application interface.

We decided that the most important principle to conserve when designing the user experience is that of the *local view* (section 4). The user must feel in full control of his local application but also fully *aware* of the activity in the collaborative group. This entails indicating group activity without distracting the user's attention from his task

The initial appearance of the application is of a traditional, single-user desktop application. One notable simplification however is the lack of a "File..." menu since storage is handled by the Joyce system.

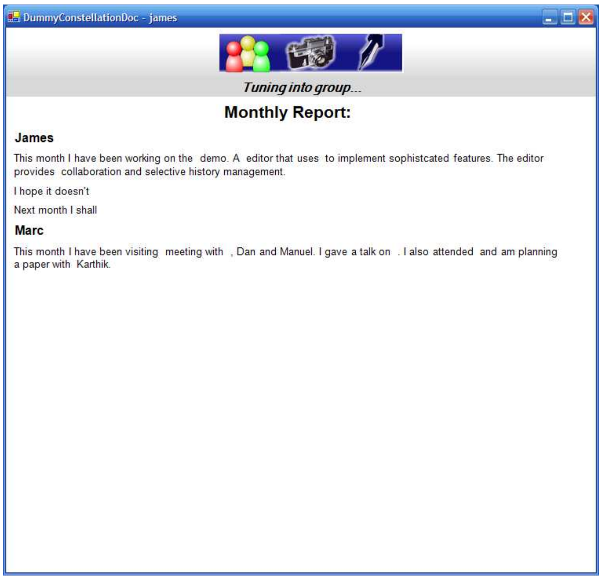

Figure 10 The start-up posture of Babble,

When ' openingà file Joyce discovers and joins the collaborative group for the document, restores the most recent snapshot it can find and brings the local multi-log up to date (section 3.6). Babble is then notified of the reconstructed state and the local interaction cycle can begin.

Figure 11 After reconstructing the state

Edits from particular collaborators can be highlighted in the text using the information in the history structure. This gives a visual projection of the hotspots in the document - the areas of the document that different collaborators are concentrating on. We can also display specific information about the contribution (taken directly from the action record).

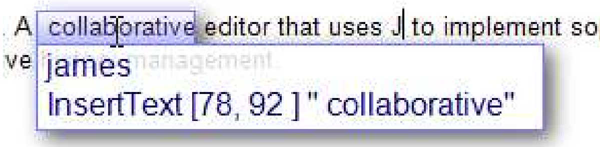

Figure 12 Tagged edits.

In keeping with the local view philosophy, the user may choose the collaborators that can contribute to his local state by instructing Joyce to filter the multi-log. Content from filtered collaborators will be removed from the display but no information is removed from the multi-log so contributions can subsequently be restored.

Conflicts are highlighted in the text using positional information from the history. For example, if a remote operation conflicts with a local one:

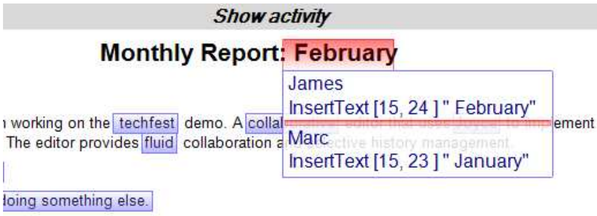

Figure 13 Viewing a conflict

Again, the user controls his local view by using the menu to instruct Joyce which action to apply to his state. If the user is the primary for the group (which in Babble is usually the creator of the document) he can instruct Joyce to treat this action as a commit/abort decision (section 3.5). For participants other than the primary, Babble uses highlighting similar to the above to indicate how far a local state has diverged from the global state.

1. History Editing

Joyce provides both a traditional, chronological view of the document history and a visual transcription of the history data-structure called the *history editor*:

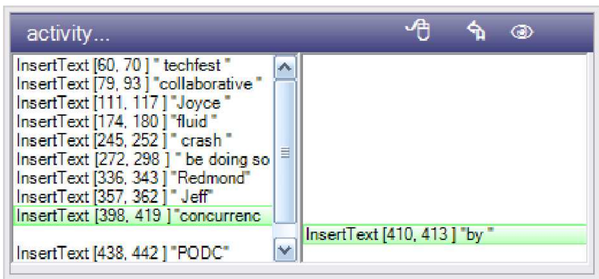

Figure 14 The history editor

Figure 14 shows the history editor. Actions are ordered according to their edit points (the character coordinate where the action began). Actions in columns other than the first column have dependencies on actions in at least one prior column, i.e. they have a strong order constraint to those actions. The selection of action *Insert "concurrency"* in the first column above also causes the dependent action *Insert "by"* in the second column to be highlighted both in the history editor and the content display, this gives the user a visual queue of the extent of a prospective undo/redo.

Since this representation can become overwhelming in large documents we can filter the displayed history according to the caret position - we call this filtering to the *local history*.

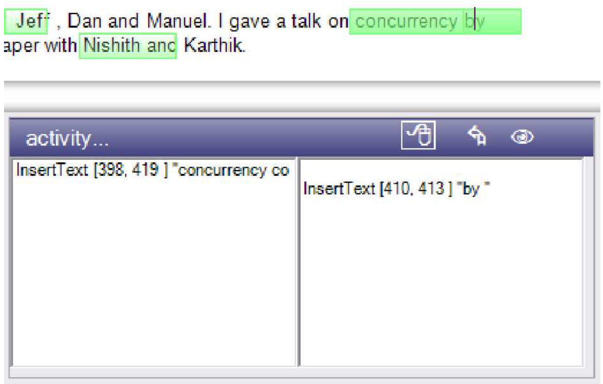

Figure 15 The local history shows the actions that occurred at the caret position.

The history editor is the interface through which the user manipulates his active subset to effect selective undo/redo. Pressing the undo button places the undo filter on the multi-log and triggers the active subset recalculation. When the result is displayed the effect is that the highlighted content modifications have been undone but the modifications of non-dependant actions remain in place. The constraints supplied to Joyce ensure the undo/redo operation is confined to only the dependent operations and the scoping mechanism ensures that we can execute the resultant active subset.

Undo/redo also applies to higher-level operations such as search and replace. In Babble search and replace is implemented as a parcel of *Delete* and *Insert* operations. These operations appear in the history editor but there is also an entry for the high-level operation. Selection of the search and replace parcel also selects its constituent operations (and any operations dependant on those). Undo/redo then works in the same way.

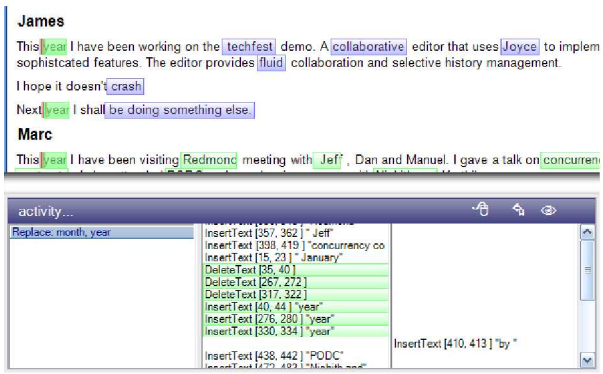

Figure 16 Selecting the search/replace constraint also selects the constituent actions.

2. Storage and Snapshots

The use of Joyce promotes a different approach to storage than that adopted in most desktop applications. Saving is no longer an explicit cognitive task for the user since the activity service automatically stores snapshots and multi-log increments. If an application instance is restarted after shutdown or a crash the Joyce system restores the state and history: in effect restoring the session.

Much has been written about the mis-match between a user's mental model of his application state and the implementation model of the filesystem [Cooper 03]. Essentially, most users have problems reconciling the idea that there are two copies of his application state, one volatile and one not, that he must sync himself.

The session restore provided by Joyce improves application usability in this area since it imparts a constancy to application state that is much closer to the user's mental model. By automating storage, session restore and history management Joyce promotes the application to the user's sole focus of attention. In Babble, an author concentrates on creating his document and lets the system take care of everything else.

Babble reserves Joyce's snapshot mechanism for when a user wishes to retain a copy of the current document state, usually because the document has reached some kind of milestone (indeed Babble uses the term milestone rather than snapshot in its interface). The interaction model is a literal visualisation: a thumbnail of the document display along with some optional meta-information about what makes this point a milestone.

Figure 17 Snapshots are represented as thumbnails.

This feature is usually approximated in current applications using the "Save As..." facility. Thanks to Joyce, Babble improves on this in several ways: the milestone does not become the active state (often a source of confusion with "Save As..."); the milestone retains the history that lead to it and, maybe most importantly, the user does not have to think of a filename - which often becomes troublesome if taking many snapshots.

• Summary and Future Work

Joyce is a programming framework that provides three main contributions: a clearly defined idea of what collaborative, nomadic applications should be, a systematic model for creating such applications and an implementation of the principles and mechanisms described in the model.

Babble demonstrates that the creation of complex, shared applications is possible with the framework. One developer was able to take the application from design to functionality in little over two months since the framework abstracted away both maintenance of occasionally-connected groups and concurrency control mechanics. The result is a full-featured, shared text editor with demonstrable advantages over similar applications: improvements in the undo/redo and storage user experience compared to contemporary single-user editors, and greater control over the local state than contemporary collaborative editors.

The creation of Babble was greatly simplified by Joyce but was still not as simple as we would have liked. Re-casting an application into Joyce's action/constraint model is difficult and requires an approach unfamiliar to most application developers. How to extensively unit test such applications remains unclear. Future work should investigate whether constraints can be automatically derived from a data type.

With regard to the programming model, strict adherence to the MVC cycle is preferable but can lead to unacceptable performance. Pure MVC implies an asynchronous model in which programs depend only on events to be notified of model changes. In reality, most MVC applications shortcut from the controller to the view to provide more immediate feedback.

In Babble there is a similar, probably typical, compromise in that local actions are constructed synchronously in the history structure and appended to the multi-log on completion. If a multi-log update arrives, special code exists to detect whether the action being constructed is *going* to conflict. If MVC is a guide this will be a typical compromise in Joyce applications; we should anticipate it and provide a lower-level API to the reconciler so that applications can detect possible conflicts themselves.

The toolkit and application described in this paper was implemented at Microsoft Research Cambridge using .NET. Our immediate focus is producing and releasing a streamlined Java version of the toolkit along with a more advanced, styled-text version of Babble and a presentation tool.

We expect further developments of the kind of application described in this paper to raise interesting and difficult questions in the areas of user-interface, application construction and security. Using Joyce, we can cope with dynamic reconfigurations of devices, users and synchrony but we can' t reconfigure an *application instance* to adapt to the device it is running on or the scenario it is being used in. An interesting approach may be to completely de-couple actions from applications. Joyce applications lessen the requirement on the user to switch mental 'modes' since his focus is always on the artefact being created. Decreasing modality increases usability. Future implementations may go further and disintegrate actions from applications completely to further lessen modality across the whole system. Actions may be associated with particular data types and always triggered in the same way. If we create a set of actions and constraints for editing XML we may be able to declaratively *generate* applications by using an XML file to weave together actions that have registered against XML schema types in a central system pool.

• Acknowledgements

The design of Joyce benefited greatly from the feedback of everyone in the distributed systems group at Microsoft Research Cambridge. The author is grateful to Beth McGreer and Dinan Gunawardena for providing invaluable feedback on early drafts of this paper.

• References

[Balasubramaniam & Pierce 1998] S. Balasubramaniam and C. Pierce: What is a File Synchronizer? *Proc. Int. Conf. On Mobile Comp. And Netw. (MobiComp 98).* ACM/IEEE, Oct. 1998

[Berlage et. al. 93] T. Berlage and A. Genau. "A Framework for Shared Applications with a Replicated Architecture". *Proc. ACM Symposium on User Interface Software and Technology*. 1993

[Cocoa] http://developer.apple.com/cocoa

[Cooper 03] A. Cooper, R. Reimann, R.M. Reimann, H. Dubberly. "About Face 2.0: The Essentials of Interaction Design". *John Wiley & Sons, Inc*. 2003

[Castro et al. 02] M. Castro, P. Druschel, A. M. Kermarrec, A. Rowstron. "SCRIBE: A large-scale and decentralized application-level multicast infrastructure". *IEEE Journal on Selected Areas in communications* (JSAC), 2002

[Demers 87] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker. HH. Sturgis, D. Swinehart, and D. Terry. "Epidemic Algorithms for Replicated Database Management". *Proc. Sixth Symposium on Principles of Distributed Computing*, Vancouver, B. C., Canada, 1987

[Edwards 97] W.K. Edwards, E. D. Mynatt, K. Petersen, M. J. Spreitzer, D. B. Terry, and M. M. Theimer. "Designing and Implementing Asynchronous Collaborative Applications with Bayou". *Proc. User Interface Systems and Technology*, Banff, Canada, 1997

[Edwards et. al. 00] W.K. Edwards, T. Igarashi, A. LaMarca, E.D. Mynatt. "A Temporal Model for Multi-Level Undo and Redo". *Proc. User Interface Systems and Technology*, San Diego, CA, 2000

[Ellis et al 88] Ellis, C., Gibbs, S.J. and Rein, G., "Design and Use of a Group Editor", *MCC Technical Report Number STP-263-88*, Sept. 1988

[Ellis et al 89] C.A. Elllis and S.J. Gibbs. "Concurrency Control in Groupware Systems". *Proc. SIGCHI Conference on Human Factors in Computing Systems*, Portland, OR, 93

[Gamma et. al. 95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. "Design Patterns, Elements of Reusable Object-Oriented Software". *Addison-Wesley*, 1995

[Greif et al. 86] I. Greif, R. Seliger, and W. Weihl (1986). "Atomic Data Abstractions in a Distributed Collaborative Editing System". Proc. of the Thirteenth Annual Symposium on Principles of Programming Languages St. Petersburg, Florida. 1986

[JotSpot] http://www.jotspot.com

[Kermarrec 01] A.M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of divergent replicas. In Proc. of Twentieth ACM Symposium on Principles of Distributed Computing PODC, Newport, RI USA, August 2001

[Krasner 88] G.E. Krasner and S.T. Pope, "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system." *Journal of Object Oriented Programming*, 1988

[Li et al. 04] D. Li and R. Li. "Ensuring Content and Intention Consistency in Real-Time Group Editors", *24th IEEE International Conference on Distributed Computing Systems* (ICDCS' 04), 2004

[McGuffin et al 92] L. McGuffin, and G. Olson, "ShrEdit: A Shared Electronic Workspace", *CSMIL Technical Report, Cognitive Science and Machine Intelligence Laboratory*, University of Michigan, 1992

[Munson et al. 96] J. Munson and P. Dewan. "A Concurrency Control Framework for Collaborative Systems". *Proc. ACM Conference on Computer Supported Cooperative Work*. 1996

[O'Brien 04] J. O'Brien and M. Shapiro, "Undo for Anyone, Anywhere, Anytime". *Proc. SIGOPS European Workshop*, 2004

[Peterson et. Al 97] K. Petersen, M.J. Spreitzer, D.B. Terry, M.M. Theimer, A.J. Demers. "Flexible Update Propagation for Weakly Consistent Replication". *Proc. Sixteenth ACM Symposium on Operating System Principles* (SOSP), Saint-Malo, Franco, 1997

[Prakash et al. 94] Prakash, A. and Shim, H. S. 1994. "DistView: support for building efficient collaborative applications using replicated objects". *Proc. ACM Conference on Computer Supported Cooperative Work* (CSCW ' 94), Chapel Hill, North Carolina, 1994).

[Preguiça et al. 03] N. Preguiça, M. Shapiro, C. Matheson: Semantics-based reconciliation for collaborative and mobile environments. In *Proc. Tenth Int. Conf. on Coop. Info. Sys. (CoopIS)*, 2003

[Roseman et al. 96] M. Roseman, S. Greenberg. "Building real-time groupware with GroupKit, a groupware toolkit". *ACM Trans. Comput.-Hum. Interact*. Mar. 1996

[Saito et al. 05] Y. Saito and M. Shapiro. "Optimistic replication". *ACM Comput. Surv. 37, 1*, 2005

[Sarin et. al. 85] S. Sarin and I. Greif. "Computer based real-time conferencing systems". *Computer 18*, 10 October 1985

[Schmidt et. al. 00] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann. "Pattern-Oriented Software Architecture Volume 2, Patterns for Concurrent and Networked Objects", *Wiley*, 2000

[Schwarz et. al. 84] P.M. Schwarz and A.Z. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems* 2, 3, 1984

[SubEtherEdit] http://www.codingmonkeys.de/subethaedit/

[Sun 02] C. Sun, "Undo as concurrent inverse in group editors", *ACM Transactions on Computer-Human Interaction* (TOCHI), 2002

[Wiki] *http://c2.com/cgi/wiki?WikiWikiWeb*